Constant-Time Snapshots with Applications to Concurrent Data Structures

Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou,
Eric Ruppert, and Yihan Sun

Concurrent Data Structures

- Important topic with applications to Database Systems, Key-Value Stores, Big Data Processing, etc
- Lots of concurrent data structures have been developed for stacks, queues, trees, etc
- Most work focuses on **single-item operations** (insert, delete, lookup, …)
- Many applications also require **multi-point queries** (range queries, snapshotting, multi-search, …)
- Lots of recent work on multi-point queries, especially range queries

Our Results

Concurrent Data Structure → Snapshottable Data Structure

Works with lots of lock-free data structures:

- Linked List **[Harris'01]**
- Queue **[MichaelScott'96]**
- BST **[EllenFatourouRuppertBreugel'10]**
- Chromatic Tree **[BrownEllenRuppert'14]**
- …

Our Results

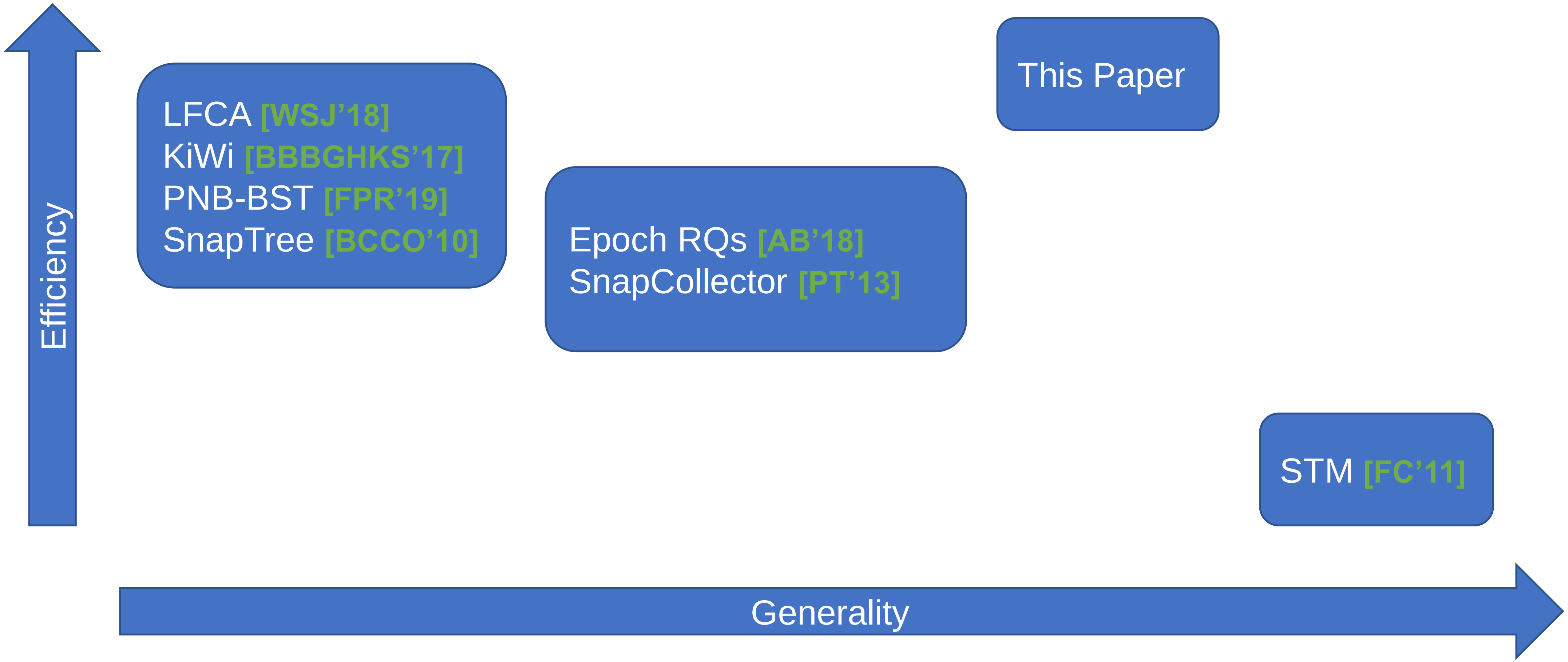
Simple, General, Efficient!

Concurrent Data Structure → Snapshottable Data Structure

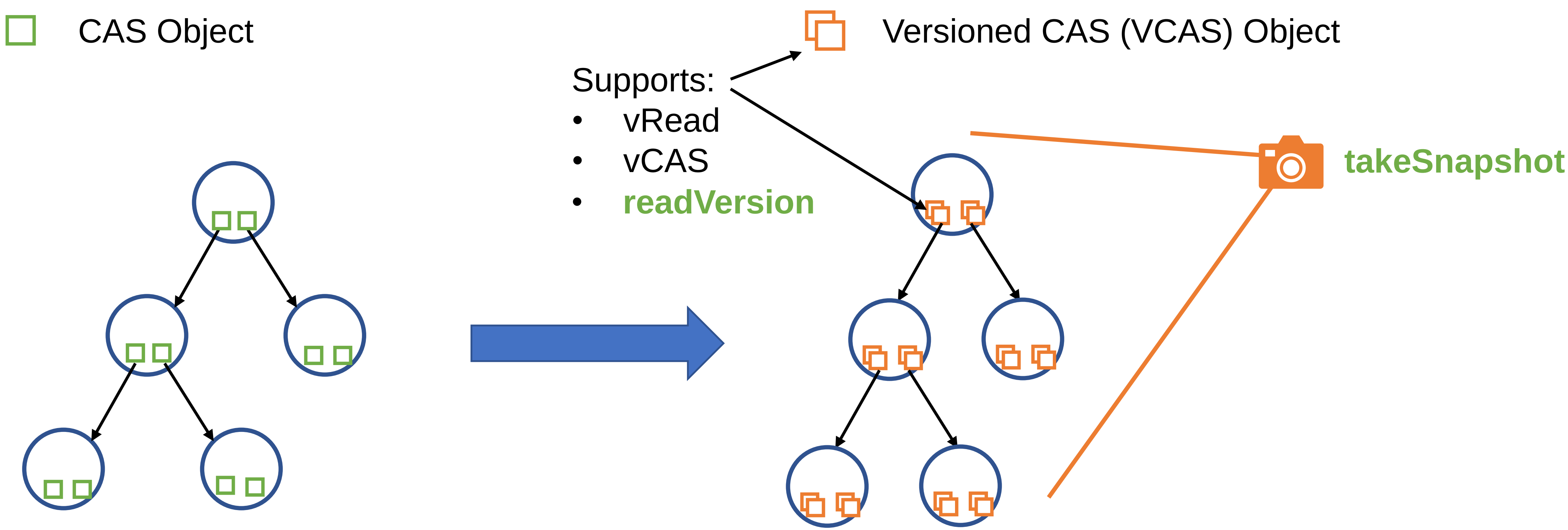
E.g. **Lock-free BST [EFRB'10]** → **Lock-free Snapshottable BST**

Lock-free BST [EFRB'10]	Lock-free Snapshottable BST	
Insert	Insert	Preserves parallelism and time bounds
Delete	Delete	
Lookup	Lookup	
	Snapshot	O(1) time, a single CAS
	Range Query	Wait-free, Linearizable
	K-Successors	
	Multi-lookup	
	Etc..	

Comparison with Existing Techniques



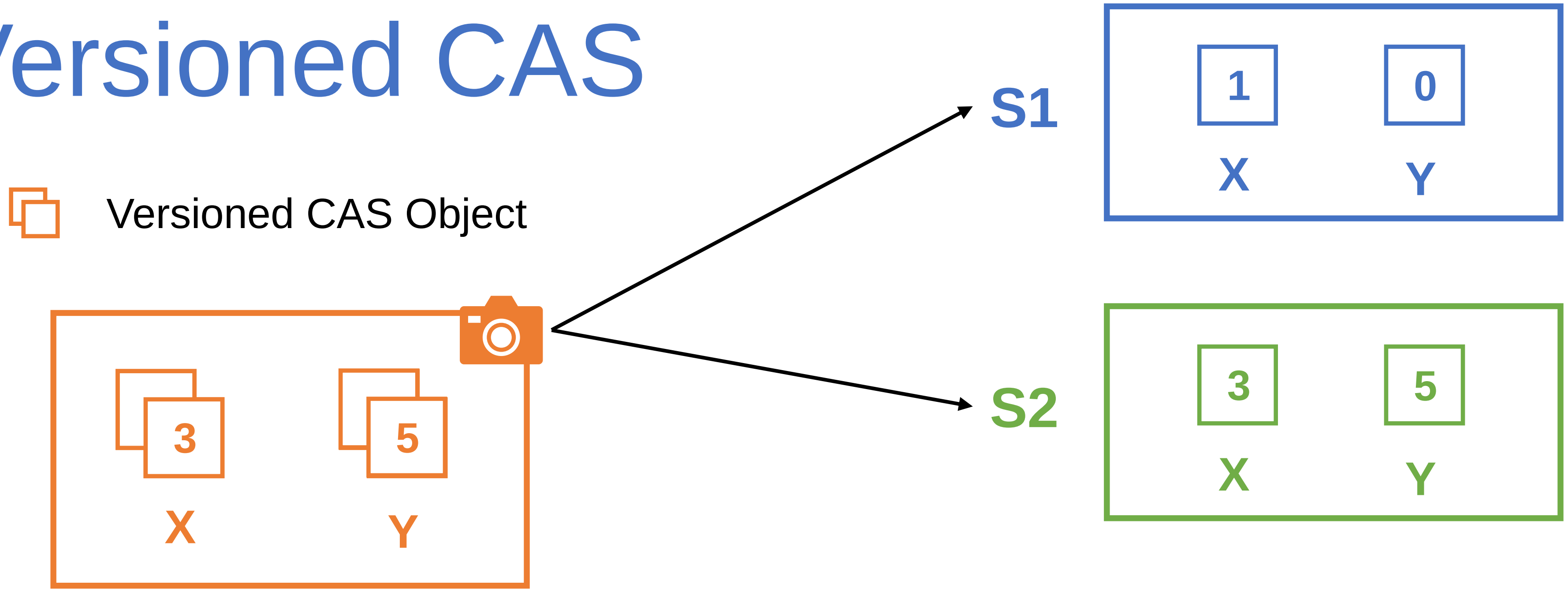
Overview of our Approach



Lock-free BST
[EFRB’10]

Lock-free Snapshottable BST

Versioned CAS



Time Complexity:

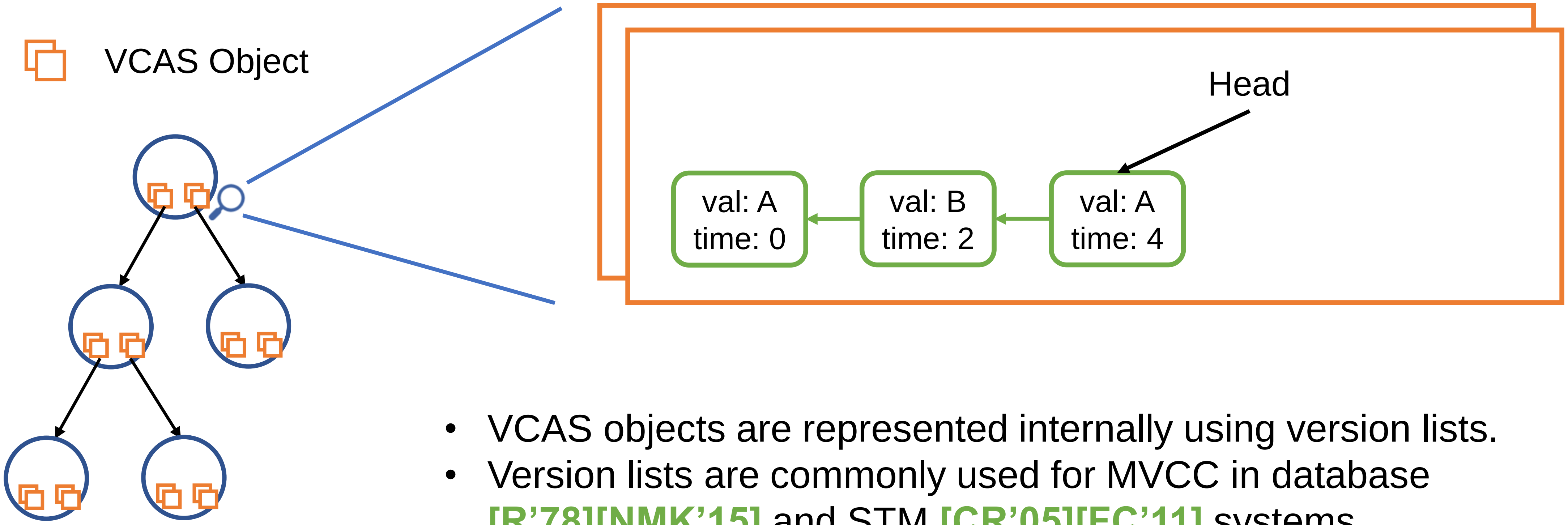
- vRead(X)
- vCAS(X, old, new)
- takeSnapshot()

O(1) time, small constant

- readVersion(X, S) — wait-free

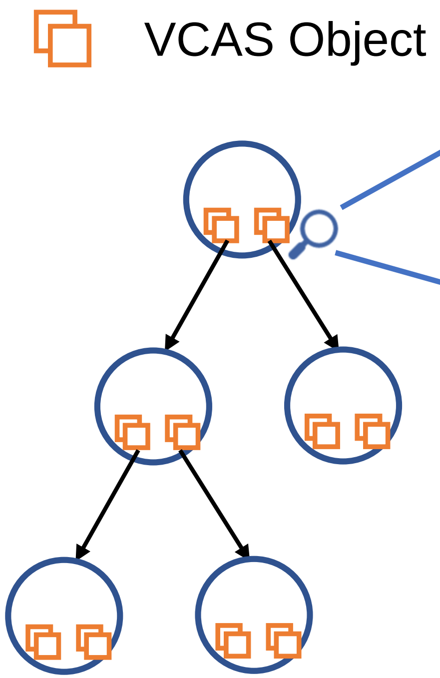
.takeSnapshot() **returns** S1
vCAS(X, 1, 3)
vCAS(Y, 0, 5)
.takeSnapshot() **returns** S2
readVersion(X, S1) **returns** 1
readVersion(Y, S2) **returns** 5

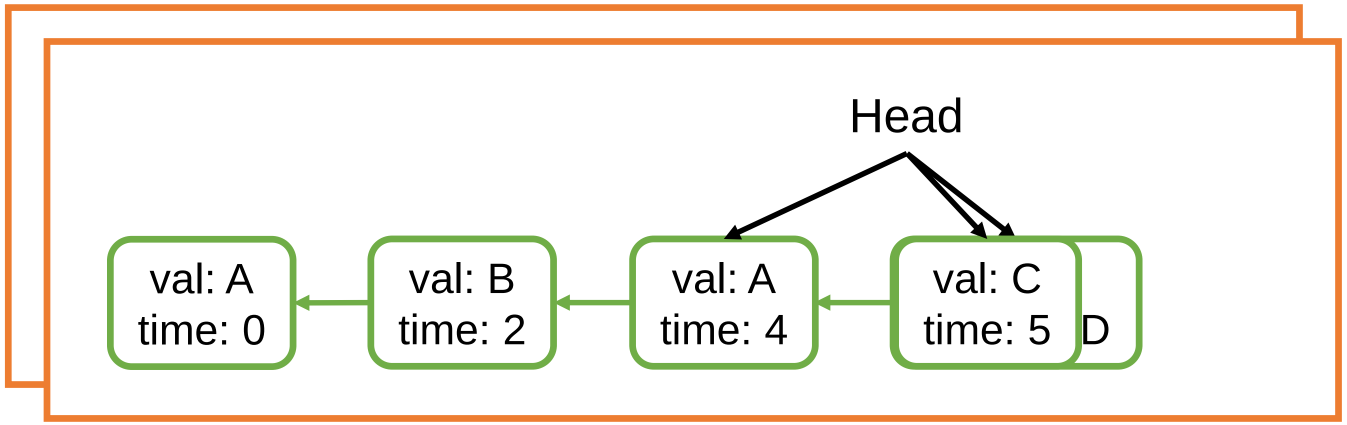
Versioned CAS Implementation



- VCAS objects are represented internally using version lists.
- Version lists are commonly used for MVCC in database **[R’78][NMK’15]** and STM **[CR’05][FC’11]** systems
- Existing lock-free implementations are expensive
- Difficulty is in atomically inserting a new version and, at the same time, assigning it an up-to-date timestamp

Versioned CAS Implementation





Current Time

6

vCAS(X, old, new)
- Link in new node with timestamp TBD
- Update its timestamp

takeSnapshot()
- Increment current time
- Return its previous value

vRead(X)
- Help update timestamp of most recent version
- Return its value

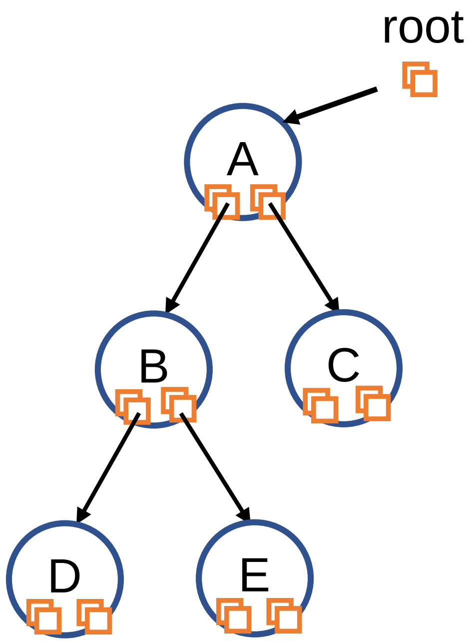
readVersion(X, *t*)
- Help update timestamp
- Find newest version with time ≤ ***t***

Versioned CAS

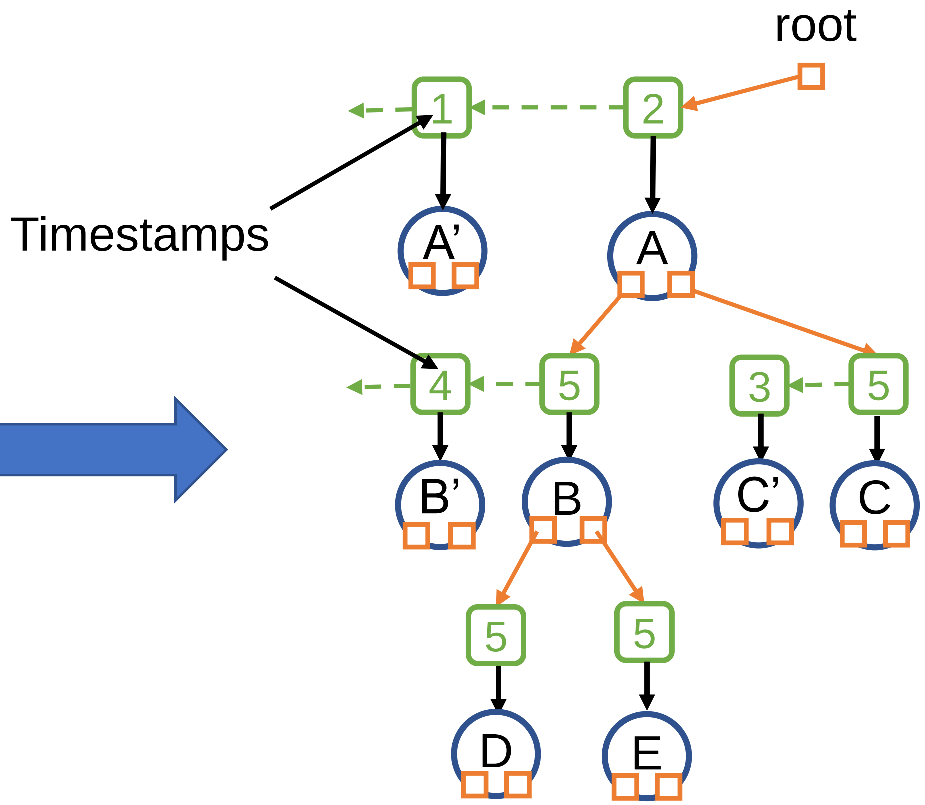
Versioned CAS Object

Version List Node

CAS Object



Snapshottable BST



Expanding out VCAS objects

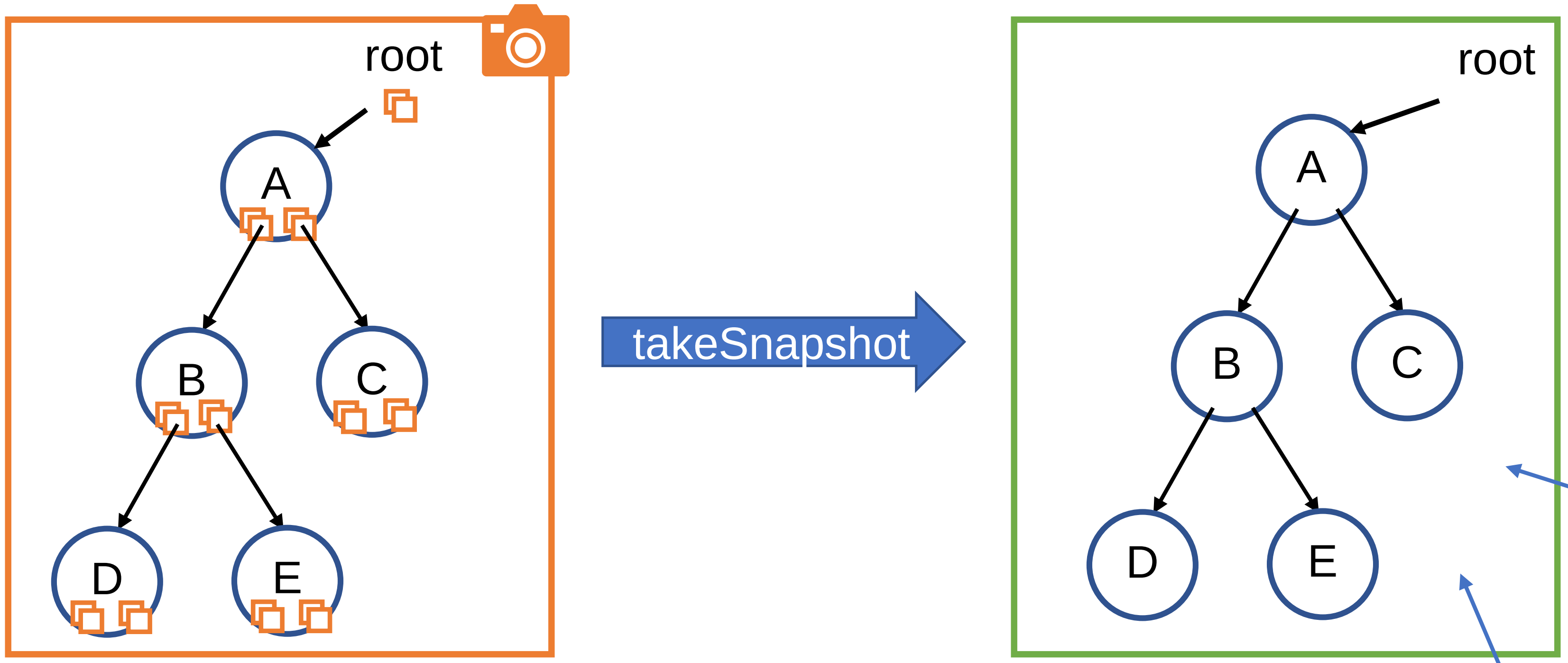
Multi-point Queries (EFRB-BST)

Versioned CAS Object

Time complexity of multi-point query: O(seq. time + contention)

# of version list nodes traversed

≤ # of inserts/deletes concurrent with the multi-point query



Snapshot accessed via readVersion()

Snapshottable BST

Sequential algorithms for rangeQuery, k-successors, etc, can be used on this snapshot.

Multi-point Queries

- In general, it is not always possible to compute linearizable multi-point queries from low-level snapshots
- In our paper, we formally define the conditions a data structure needs to satisfy for this approach to work
- Most lock-free data structures satisfy this condition

Our Results

Concurrent Data Structure → Snapshottable Data Structure

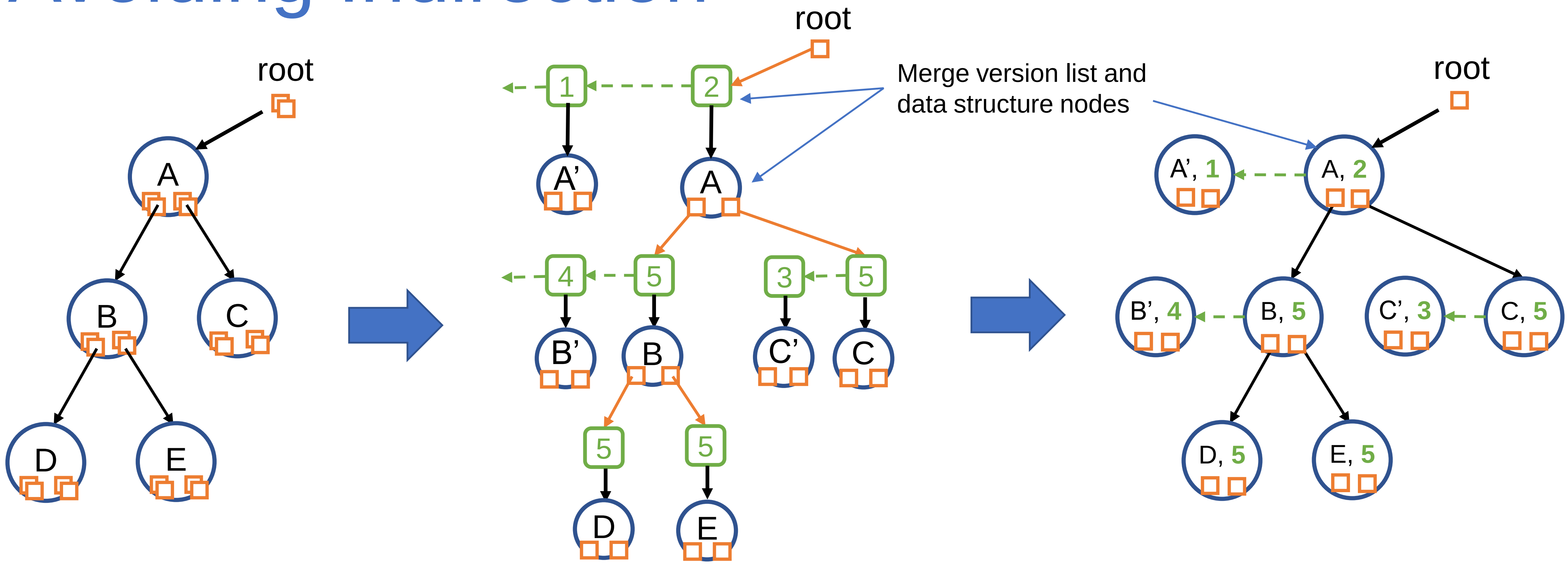
E.g. **Lock-free BST [EFRB'10]** → **Lock-free Snapshottable BST**

Lock-free BST [EFRB'10]	Lock-free Snapshottable BST	
Insert	Insert	Preserves parallelism and time bounds
Delete	Delete	
Lookup	Lookup	
	Snapshot	O(1) time, a single CAS
	Range Query	Wait-free, Linearizable
	K-Successors	
	Multi-lookup	
	Etc..	

Practical Optimizations

- Avoiding Indirection
- Using exponential backoff to reduce contention on global timestamp
- Removing redundant versions from the version list
- Garbage collecting old versions
- Our experiments include all the above optimizations

Avoiding Indirection



Snapshottable BST

Expanding out VCAS objects

Without indirection

Experimental Evaluation

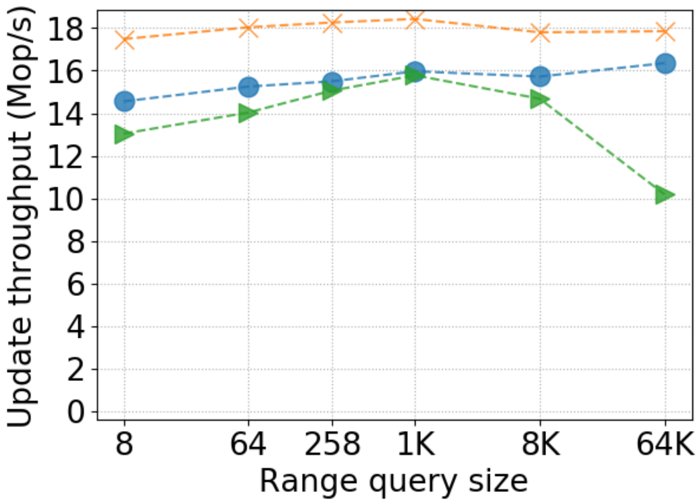
- Goals:
  1. Understand how much overhead our approach adds to the original data structure
  2. Compare performance of our approach with state-of-the-art range queryable data structures
- Machine: 72-core (4-socket) Intel machine with 2-way hyperthreading
- Evaluated performance in both C++ and Java

Experimental Evaluation (C++)

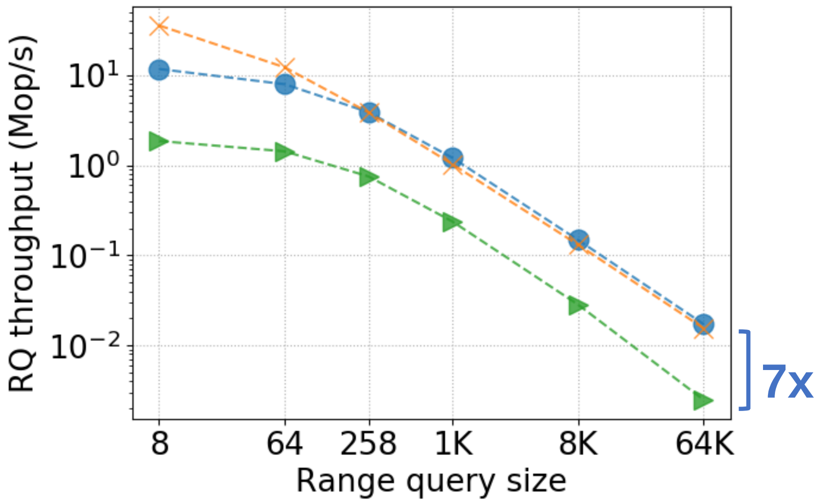
BST with **non-atomic** range queries **[AB’18]**

BST with **atomic** range queries **[AB’18]**

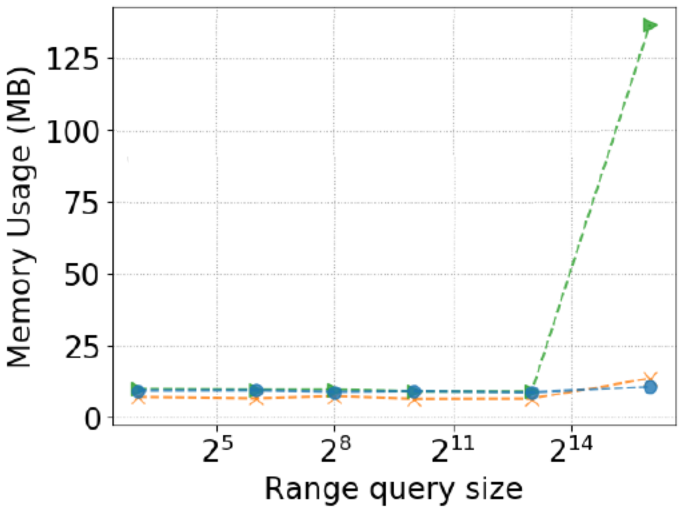
BST using Versioned CAS objects to support **atomic** range queries



(A) Update Throughput



(B) Range Query Throughput
(log scale)



(C) Memory Usage

Workload: 36 update threads, 36 RQ threads, run on a tree of size 100,000

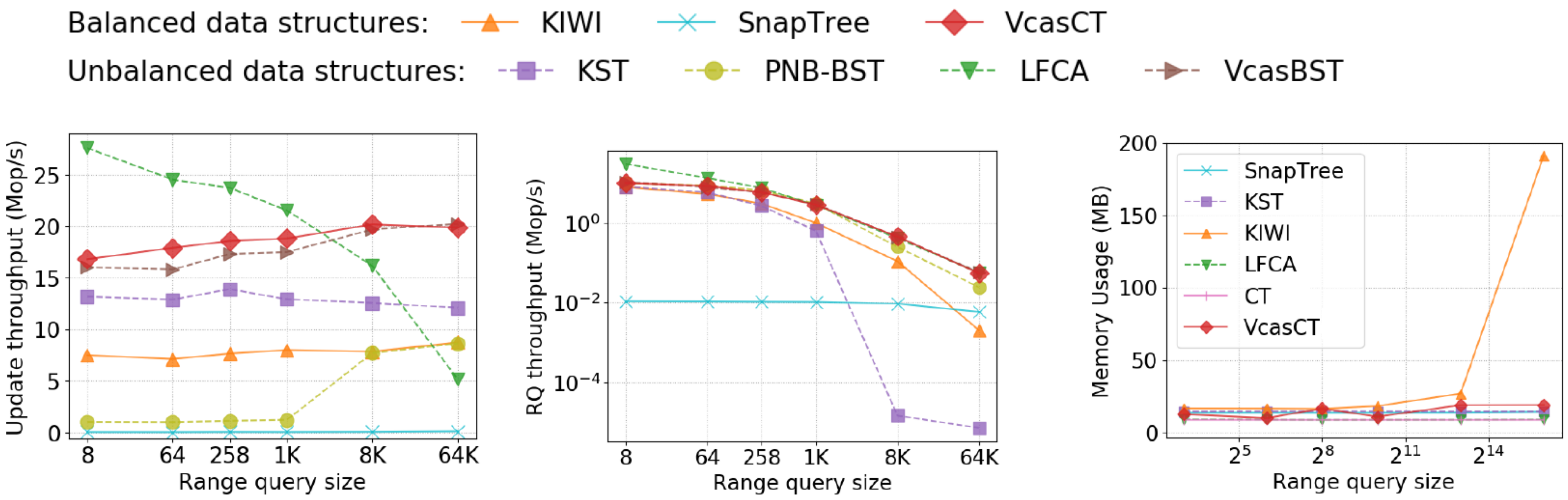
Experimental Evaluation (Java)

- Data Structures:
  - PNB-BST [FPR'19] – Persistent BST supporting range queries
  - **KiWi** [BBBGHKS'17] – Key-Value store supporting range queries
  - LFCA [WSJ'18] – Lock-free Contention Adapting Search Tree
  - KST [BH'11] – K-ary Search Tree
  - **SnapTree** [BCCO'10] – Lock-based BST supporting snapshots

* Data structures in blue are **balanced**

- Our Implementations:
  - VcasBST – snapshottable version of EFRB-BST [EFRB'10]
  - **VcasCT** – snapshottable version of **ChromaticTree** [BER'14]

Experimental Evaluation (Java)



(A) Update Throughput

(B) Range Query Throughput
(log scale)

(C) Memory Usage

Workload: 36 update threads, 36 RQ threads, run on a tree of size 100,000.

Experimental Evaluation (Summary)

- More experiments/workloads can be found in our paper
- Overall, we find that our approach adds very little overhead to the original data structure
- Furthermore, our general-purpose approach is often as fast as, or faster than, state-of-the-art lock-free structures supporting range queries.

Conclusion

- We presented an approach for adding snapshotting and multi-point queries to existing concurrent data structures
  - **Easy-to-use**: simply replace CAS with Versioned CAS
  - **Efficient**: both theoretically and practically
  - **General**: supports a wide range of data structures and multi-point queries

- Our code is available on GitHub: https://github.com/yuanhaow/vcaslib